IMPROVED UPDATE/QUERY ALGORITHMS FOR THE INTERVAL VALUATION PROBLEM

D.S. Hirschberg and D.J. Volper
University of California, Irvine

4 June 1986

Key words: query, range tree.

Abstract

Let I be the set of intervals with end points in the integers $1 \ldots n$. Associated with each element in I is a value from a commutative semigroup S. Two operations are to be implemented: update of the value associated with an interval and query of the sum of the values associated with the intervals which include an integer.

If the values are from a commutative group (i.e., inverses exist) then there is a data structure which enables both update and query algorithms of time complexity $O(\log n)$. For the semigroup problem, the use of range trees enables both update and query algorithms of time complexity $O(\log^2 n)$.

Data structures are presented for the semigroup problem with (update,query) algorithms of complexities $(\log n, \log^2 n)$, $(\log n \log\log n, \log n)$.

Introduction

Let I be the set of intervals with end points in the integers $1 \ldots n$. Associated with each element in I is a value from a commutative semigroup S. Let *sum* refer to the semigroup operator. Examples of such operators over the integers are addition, multiplication and minimum. Two operations are to be implemented, update and query. An update of an interval changes the value associated with that interval. A query of integer k returns the sum of values associated with the intervals which include k.

We consider solutions of the class that involves variables storing values in S. Each variable stores the sum of values associated with some subset of I. A query is answered by summing a subset of these variables. An update is accomplished by recomputing the values of the appropriate variables. This class has been used by Fredman and others [F80, BFK81, F81, F81b] for analysis of query problems.

If the values associated with the interval are from a commutative group (i.e., inverses exist) then there is a data structure which enables both update and query algorithms of time complexity $O(\log n)$ [F79]. For the semigroup problem, the use of range trees enables both update and query algorithms of time complexity $O(\log^2 n)$ [W78, L78, LW82, W85].

We present data structures with update and query algorithms of the following complexities.

Update Time	Query Time
$O(\log n)$	$O(\log^2 n)$
$O(\log n \log\log n)$	$O(\log n)$

Table 1

Data Structure and Algorithm

We associate each interval $[i, j]$ with a point in a two dimensional plane whose horizontal (x) and vertical (y) coordinates are i and j, the values of the endpoints of the interval, respectively. These points lie within the upper left triangular region of an n by n square. A query of k needs to retrieve the sum of the values associated with all points whose x coordinate is $\leq k$ and whose y coordinate is $\geq k$. That is, the query region consists of all points which lie in a rectangle whose upper left corner is at $(1, n)$ and whose lower right corner is at (k, k), as illustrated in Figure 1.

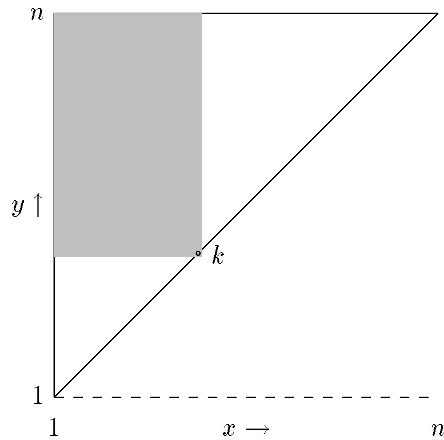

Figure 1 — Query Region of k

To clarify membership of a rectangle we use the convention that rectangle boundaries are midway between points, and thus no points can occur on the boundary of a rectangle. Variables may be represented by a region on the graph which covers the set of points corresponding to the intervals whose values are included in the variable. A query may be answered by summing variables whose regions form a disjoint cover of the rectangle associated with the query. Updating an interval involves recomputing the value of each variable whose region includes the point associated with the interval.

For ease of explanation, we assume that n is $2^m - 2$. We partition the triangular region (which corresponds to the universe of intervals) into L-shaped subregions. Details of a single L-shaped subregion are shown in Figure 6, while a partition using these L-shaped subregions is given in Figure 2. Each subregion on the diagonal includes exactly three intervals $[k, k]$, $[k+1, k+1]$, and $[k, k+1]$, for odd k. Note that there are $2^{m-1} - 1$ L's on the diagonal, each having unit width. Behind each diagonal of L's is another diagonal of L's having the same shape and orientation but of twice the size. There are $2^{m-1-i} - 1$ L's of width 2^i. In Figure 2, there are 7 L's of unit width, 3 L's of width 2, and one L of width 4.

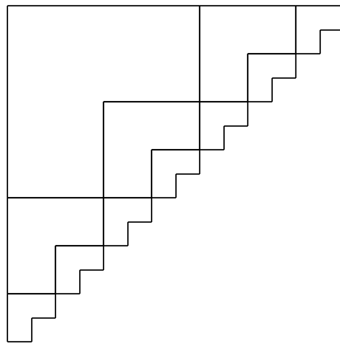

Figure 2 — Partition of Interval Universe into L's

An aggregate is a rectangular region that begins from a leg of an L region, has the width of that L, and extends through that L across the universal region. There is an aggregate in both the horizontal and vertical directions for each L. Shown below is a vertical aggregate of size 1 and one of size 2. There is a similar set of aggregates with orientations in the horizontal direction. For our diagram below there would be a total of 14 aggregates of width 1, 6 of width 2 and 2 of width 4.

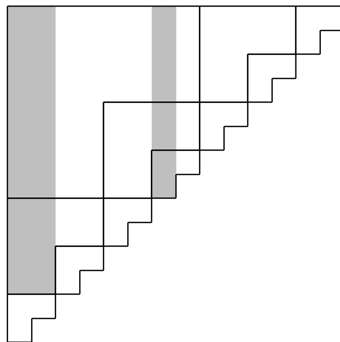

Figure 3 — Sample aggregates

Any query region may be covered by an appropriately chosen set of aggregates. These are chosen as follows. Given a query k, it lies in a small L. Select the aggregate of width 1 associated with that L and having the proper orientation. There remains to be covered a rectangular region whose lower left corner is in an L of size 2. Select the aggregate of width 2 associated with that L and having the proper orientation. There now remains a rectangle whose corner is in an L of size 4. Such a selection is shown in Figure 4. Any rectangle can be disjointly covered by such a recursive selection using $\log n$ aggregates.

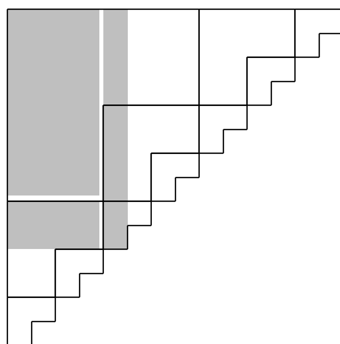

Figure 4 — Covering a query with 3 aggregates

A point is in at most one horizontal and one vertical aggregate of each size. Thus the value of a point affects $O(\log n)$ aggregates. Each query uses at most one aggregate of each size, that is $O(\log n)$ aggregates. The update and query times are $O(\log n)$ multiplied by the time to, respectively, update and query an aggregate.

If inverses exist within the semigroup, we can utilize the aggregates themselves as our variables. A modification of the value associated with an interval results in a similar modification of the variables associated with the appropriate aggregates. This is achieved by subtracting the interval's old value and adding its new value to these aggregates. A query is achieved by summing the variables associated with appropriate aggregates. This yields a structure which, like that of [F79], has both query and update times of $O(\log n)$.

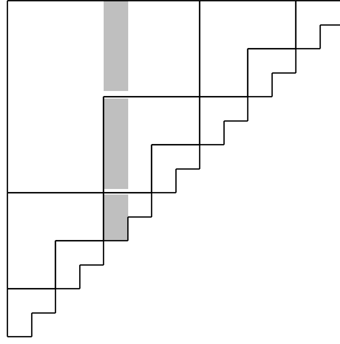

Figure 5 — Splitting an aggregate at the L boundaries

For the general case (with no inverses) we must supply additional structure. We partition the aggregates into *parts* by splitting them where ever they cross the boundary of an L. Figure 5 illustrates the three parts of one of the aggregates from Figure 4. Because the size of the L's grow exponentially, an aggregate is split into at most $\log n$ parts. Queries will use the parts of each aggregate to compute the sum of the values within the aggregate. An update now affects only the parts within a single L, and there are $O(\log n)$ such parts. We develop a structure within each L to permit all these parts to be updated in $O(\log n)$ time regardless of the size of the L.

We divide each of the L's into overlapping rectangular regions which we will call *strips*. There are vertical as well as horizontal strips.

We describe vertical strips which are shown in Figure 6; horizontal strips are symmetric in description. The strips have width 1. A point lies in exactly one vertical strip. The points within each strip are organized into range query tree, which is a binary tree with the leaves holding the values of the points and each internal node holds the sum of the values of the leaves in its subtree. The root of the tree holds the value of the strip (sum of the values of all points within the strip). Thus, we can update the value of a strip by recomputing the sums along the path from the updated point to the root in time $O(\log n)$. The vertical strips themselves, or more particularly the variables associated with the root of the range tree within each strip, are also organized into a range tree, called the L range tree. After a strip has been updated, the L range tree can be updated in $O(\log n)$. We observe that the part associated with an aggregate of width 1 is a strip and so is a leaf in this range tree; the part associated with an aggregate of width 2 is the combination of two adjacent strips and so forth. Thus, the nodes in the L range tree contain the values of the parts within the L. Therefore, in

response to a query, the value associated with any part can be delivered in constant time by accessing the appropriate node in the L range tree.

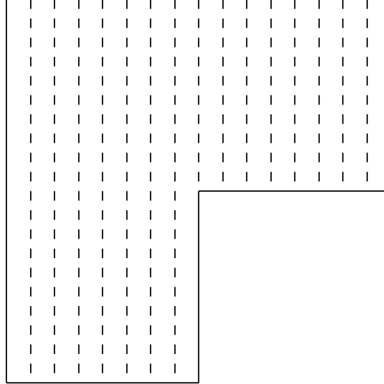

Figure 6 — Strips within an L of size 16

We now present two structures based upon the above construction. The parts of each aggregate can be formed into a range query structure called the aggregate range structure. The only value of interest for a query is that of the aggregate, that is, the sum of all the parts. The aggregate range structure has $O(\log n)$ parts. If we use a binary tree for this structure with the parts as the leaves and with each internal node containing the sum of its children, retrievals can be performed by accessing the root, while updates are performed by recomputing sums along a path from the part to the root. Accessing the root is trivially an $O(1)$ operation. Recomputing the path from a part to the root is proportional to the height of the tree. Assuming we chose to use some balanced form of binary tree, this height is the logarithm of the number of leaves, that is, $\log(O(\log n))$. If we use an array whose elements are the values of the parts, updates affect one element while retrievals are performed by forming the sum of the elements. In this case updates are $O(1)$ while the cost of a retrieval is proportional to the number of elements in the array, that is, $O(\log n)$. Using these two structures, we obtain the following times for queries of an aggregate range structure.

Structure	Update Time	Query Time
Array	$O(1)$	$O(\log n)$
Tree	$O(\log \log n)$	$O(1)$

Table 2.

As previously discussed, the update and query times are $O(\log n)$ multiplied by the update and query times for the aggregate range structure plus, in the case of an update, an additional $O(\log n)$ to update the strip and L range trees. This yields the results in Table 1. Finally, we note that we could obtain intermediate values for Table 2 using the structures presented in [F82].

References

[BFK81] Burkhard, Walter. A., Michael L. Fredman, and Daniel J. Kleitman, Inherent complexity trade-offs for range query problems. *Theoretical Computer Science*, **16**, 1981, 279-290.

[F82] Fredman, Michael L., The complexity of maintaining an array and computing its partial sums. *Journal of the Association of Computing Machinery*, **29:1**, January 1982, 250-260.

[F81] Fredman, Michael L., Lower bounds on the complexity of some optimal data structures. *SIAM Journal on Computing*, **10**, 1981, 1-10.

[F80] Fredman, Michael L., The inherent complexity of dynamic data structures which accomodate range queries. *Proceedings of the 1980 IEEE Symposium on Foundations of Computer Science*, 191-199.

[F79] Fredman, Michael L., A near optimal data structure for a type of range query problem. *Proceedings of the 11th Annual Symposium on the Theory of Computer Science*, 1979, 62-66.

[F81b] Fredman, Michael L., The spanning bound as a measure of range query complexity. *Journal of Algorithms*, **2**, 1981, 77-87.

[L78] Lueker, George S., A data structure for dynamic range queries. *Proceedings of the 1978 IEEE Symposium on Foundations of Computer Science.*

[LW82] Lueker, George S., and Dan E. Willard, A data structure for dynamic range queries. *Information Processing Letters*, **15:15**, December 1982, 209-213.

[W78] Willard, Dan E., New data structures for orthogonal queries. Technical Report TR-22-78, Center for Research in Computing Technology, Harvard University.

[W85] Willard, Dan E., New data structures for orthogonal queries. *SIAM Journal of Computing*, **14:1**, February 1985, 232-253.